Logs, Logs, Logs!

Welcome to Logs, Logs, Logs! on Exercism's Go Track. If you need help running the tests or submitting your code, check out `HELP.md`. If you get stuck on the exercise, check out `HINTS.md`, but try and solve it without using those first :)

Introduction

The `rune` type in Go is an alias for `int32`. Given this underlying `int32` type, the `rune` type holds a signed 32-bit integer value. However, unlike an `int32` type, the integer value stored in a `rune` type represents a single Unicode character.

Unicode and Unicode Code Points

Unicode is a superset of ASCII that represents characters by assigning a unique number to every character. This unique number is called a Unicode code point. Unicode aims to represent all the world's characters including various alphabets, numbers, symbols, and even emoji as Unicode code points.

In Go, the `rune` type represents a single Unicode code point.

The following table contains example Unicode characters along with their Unicode code point and decimal values:

Unicode Character	Unicode Code Point	Decimal Value
0	`U+0030`	`48`
A	`U+0041`	`65`
a	`U+0061`	`97`
¿	`U+00BF`	`191`
π	`U+03C0`	`960`
🧠	`U+1F9E0`	`129504`

UTF-8

UTF-8 is a variable-width character encoding that is used to encode every Unicode code point as 1, 2, 3, or 4 bytes. Since a Unicode code point can be encoded as a maximum of 4 bytes, the `rune` type needs to be able to hold up to 4 bytes of data. That is why the `rune` type is an alias for `int32` as an `int32` type is capable of holding up to 4 bytes of data.

Go source code files are encoded using UTF-8.

Using Runes

Variables of type `rune` are declared by placing a character inside single quotes:

```
myRune := '¿'
```

Since `rune` is just an alias for `int32`, printing a rune's type will yield `int32`:

```
myRune := '¿'
fmt.Printf("myRune type: %T\n", myRune)
// Output: myRune type: int32
```

Similarly, printing a rune's value will yield its integer (decimal) value:

```
myRune := '¿'
fmt.Printf("myRune value: %v\n", myRune)
// Output: myRune value: 191
```

To print the Unicode character represented by the rune, use the `%c` formatting verb:

```
myRune := '¿'
fmt.Printf("myRune Unicode character: %c\n", myRune)
// Output: myRune Unicode character: ¿
```

To print the Unicode code point represented by the rune, use the `%U` formatting verb:

```
myRune := '¿'
fmt.Printf("myRune Unicode code point: %U\n", myRune)
```

```
// Output: myRune Unicode code point: U+00BF
```

Runes and Strings

Strings in Go are encoded using UTF-8 which means they contain Unicode characters. Since the `rune` type represents a Unicode character, a string in Go is often referred to as a sequence of runes. However, runes are stored as 1, 2, 3, or 4 bytes depending on the character. Due to this, strings are really just a sequence of bytes. In Go, slices are used to represent sequences and these slices can be iterated over using `range` .

Even though a string is just a slice of bytes, the `range` keyword iterates over a string's runes, not its bytes. In this example, the `index` variable represents the starting index of the current rune's byte sequence and the `char` variable represents the current rune:

```
myString := "❗hello"
for index, char := range myString {
  fmt.Printf("Index: %d\tCharacter: %c\t\tCode Point: %U\n", index, char, char)
}
// Output:
// Index: 0     Character: ❗           Code Point: U+2757
// Index: 3     Character: h            Code Point: U+0068
// Index: 4     Character: e            Code Point: U+0065
// Index: 5     Character: l            Code Point: U+006C
// Index: 6     Character: l            Code Point: U+006C
// Index: 7     Character: o            Code Point: U+006F
```

Since runes can be stored as 1, 2, 3, or 4 bytes, the length of a string may not always equal the number of characters in the string. Use the builtin `len` function to get the length of a string in bytes and the `utf8.RuneCountInString` function to get the number of runes in a string:

```
import "unicode/utf8"

myString := "❗hello"
stringLength := len(myString)
numberOfRunes := utf8.RuneCountInString(myString)

fmt.Printf("myString - Length: %d - Runes: %d\n", stringLength, numberOfRunes)
// Output: myString - Length: 8 - Runes: 6
```

Instructions

You have been tasked with creating a log library to assist with managing your organization's logs. This library will allow users to identify which application emitted a given log, to fix corrupted logs, and to determine if a given log line is within a certain character limit.

1. Identify which application emitted a log

Logs come from multiple applications that each use their own proprietary log format. The application emitting a log must be identified before it can be stored in a log aggregation system.

Implement the `Application` function that takes a log line and returns the application that emitted the log line.

To identify which application emitted a given log line, search the log line for a specific character as specified by the following table:

Application	Character	Unicode Code Point
`recommendation`	❗	`U+2757`
`search`	🔍	`U+1F50D`
`weather`	☀	`U+2600`

If a log line does not contain one of the characters from the above table, return `default` to the caller. If a log line contains more than one character in the above table, return the application corresponding to the first character found in the log line starting from left to right.

```
Application("❗ recommended search product 🔍")
// => recommendation
```

2. Fix corrupted logs

Due to a rare but persistent bug in the logging infrastructure, certain characters in logs can become corrupted. After spending time identifying the corrupted characters and their original value, you decide to update the log library to assist in fixing corrupted logs.

Implement the `Replace` function that takes a log line, a corrupted character, and the original value and returns a modified log line that has all occurrences of the corrupted character replaced with the original value.

```
log := "please replace '👎' with '👍'"

Replace(log, '👎', '👍')
// => please replace '👍' with '👍'"
```

3. Determine if a log can be displayed

Systems responsible for displaying logs have a limit on the number of characters that can be displayed per log line. As such, users are asking for this library to include a helper function to determine whether or not a log line is within a specific character limit.

Implement the `WithinLimit` function that takes a log line and character limit and returns whether or not the log line is within the character limit.

```
WithinLimit("hello❗", 6)
// => true
```

Source

Created by

- @sudomateo
- @tehsphinx